On-demand and Parallel Checkpoint/Restore for GPU Applications

Yanning Yang[1,2], Dong Du[1,2]✉, Haitao Song[3], Yubin Xia[1,2]

[1]Institute of Parallel and Distributed Systems, SEIEE, Shanghai Jiao Tong University

[2]Engineering Research Center for Domain-specific Operating Systems, Ministry of Education

[3]Shanghai Artificial Intelligence Research Institute

Shanghai, China

ABSTRACT

Leveraging serverless computing for cloud-based machine learning services is on the rise, promising cost-efficiency and flexibility are crucial for ML applications relying on high-performance GPUs and substantial memory. However, despite modern serverless platforms handling diverse devices like GPUs seamlessly on a pay-as-you-go basis, a longstanding challenge remains: startup latency, a well-studied issue when serverless is CPU-centric. For example, initializing GPU apps with minor GPU models, like MobileNet, demands several seconds. For more intricate models such as GPT-2, startup latency can escalate to around 10 seconds, vastly overshadowing the short computation time for GPU-based inference. Prior solutions tailored for CPU serverless setups, like fork() and Checkpoint/Restore, cannot be directly and effectively applied due to differences between CPUs and GPUs.

This paper presents gCROP (GPU Checkpoint/Restore made On-demand and Parallel), the first GPU runtime that achieves <100ms startup latency for GPU apps with up to 774 million parameters (3.1GB GPT-2-Large model). The key insight behind gCROP is to selectively restore essential states *on demand* and *in parallel* during boot from a prepared checkpoint image. To this end, gCROP first introduces a global service, GPU Restore Server, which can break the existing barrier between restore stages and achieve parallel restore. Besides, gCROP leverages both CPU and GPU page faults, and can on-demand restore both CPU and GPU data with profile-guided order to mitigate costs caused by faults. Moreover, gCROP designs a multi-checkpoint mechanism to increase the common contents among checkpoint images and utilizes deduplication to reduce storage costs. Implementation and evaluations on AMD GPUs show significant improvement in startup latency, 6.4x–24.7x compared with booting from scratch and 3.9x–23.5x over the state-of-the-art method (CRIU).

CCS CONCEPTS

• **Computer systems organization → Cloud computing**.

KEYWORDS

Cloud Computing; Startup Latency; Checkpoint and Restore; GPUs

ACM Reference Format:

Yanning Yang, Dong Du, Haitao Song, Yubin Xia. 2024. On-demand and Parallel Checkpoint/Restore for GPU Applications. In *ACM Symposium on Cloud Computing (SoCC '24), November 20–22, 2024, Redmond, WA, USA.* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3698038.3698510

1 INTRODUCTION

With the increasing popularity of AI/ML apps, e.g., ChatGPT [17], Sora [27], and image recognition, machine learning has been widely adopted in today's cloud. Cloud tenants [5, 59, 65] can deploy their own machine learning models, providing diverse AI services to users. However, the substantial computing demands of these AI services (mostly model *inference*) cannot be efficiently met by CPUs alone. To address the challenge and enhance AI service performance, cloud vendors tend to deploy and support GPUs on the cloud for developers and end-users to use.

Compared with (cheaper) CPU instances, GPUs bring significant resource and cost challenges. On AWS, a GPU instance (p3.2xlarge with one V100 [19] GPU, price: 1524.24

SoCC '24, November 20–22, 2024, Redmond, WA, USA

ACM ISBN 979-8-4007-1286-9/24/11…$15.00
https://doi.org/10.1145/3698038.3698510

✉Corresponding author: Dong Du (dd_nirvana@sjtu.edu.cn).

USD per month) is about 9x expensive compared with CPU instances (t2.2xlarge) [7]. To mitigate the costs and also fit the on-demand nature of services, a recent trend to deploy GPU apps are utilizing serverless computing, which promises cost-efficiency and flexibility, e.g., ServerlessLLM [44].

However, despite modern serverless platforms handling diverse devices like GPUs seamlessly on a pay-as-you-go basis (e.g., AWS SageMaker [65] and Alibaba serverless GPU [5]), a longstanding challenge remains: *long startup latency*. For example, initializing GPU apps with small GPU models, like MobileNet, demands several seconds (§2.2). For more intricate models such as GPT-2, startup latency can escalate to around 10 seconds, vastly overshadowing the short computation time for serverless tasks like GPU-based inference (e.g., ~5ms – ~500ms). Although cold startup latency is a well-studied problem for CPU-centric serverless computing [56], it has little progress for GPU apps. Prior solutions [26, 31, 42, 61, 69, 71, 72] tailored for CPU serverless setups, e.g., `fork()`, cannot be directly and effectively applied due to the lack of some features in GPUs. For example, GPUs do not have the OS kernel which can help to reuse application states with `fork()`.

State-of-the-art efforts have explored caching [76, 78, 80], API forwarding [43, 60, 79], and optimized loading [25, 44, 81], to optimize the startup latency. However, the effectiveness and efficiency of caching highly rely on the policies, and it may incur non-trivial resource overheads to keep GPU apps alive. Although API forwarding (e.g., FaaSwap [79]) can skip the GPU context and library initialization costs (about 800ms for a PyTorch application on our AMD platform), it introduces inter-process communication (IPC) overhead between GPU apps and the GPU server. Optimized loading systems, e.g., ServerlessLLM [44] and SAGE [81], design better image formats and utilize OS and hardware features (e.g., `mmap()`) to improve the loading performance. However, the latency is still far from ideal — the actual execution time is usually less than 10% of the startup time (§2.2).

This paper presents gCROP, the first GPU runtime system that achieves <100ms startup latency for large GPU apps (GPT-2-Large). gCROP will manage the GPU apps in the local machine, and can be incorporated with serverless systems like prior container-based serverless runtimes (e.g., runc [22] and gVisor [11]). gCROP adopts the idea of *init-less booting* [42], skipping the initialization with checkpoint images, which has been widely explored in CPU-centric serverless systems. Specifically, gCROP leverages Checkpoint/Restore which can directly restore from a well-prepared checkpoint image, therefore skipping many unnecessary (and redundant) initialization tasks.

Designing an efficient Checkpoint/Restore (C/R) for GPU-based serverless apps is non-trivial — our experiments on CRIU [9] show that it still has 193ms–2,278ms startup latency (image cached in memory). Specifically, gCROP needs to resolve three major challenges. First, the restore procedure for GPU-based apps is significantly longer compared with CPU apps. Besides the CPU data in DRAM, restore for GPU apps still needs to load the GPU data including model parameters to the GPU (VRAM), taking several seconds for large models. An intuitive idea is to parallelize the CPU and GPU recovery, however, we observe the existence of *restorer barrier*, which is the root cause of why existing GPU C/R can only perform (mostly) in a sequential way — GPU data need to be restored before the barrier while CPU data need to be restored after the barrier (§2.3). Second, GPUs complicate the on-demand recovery. When optimizing C/R for CPU processes, a common strategy involves utilizing `mmap()` for on-demand recovery [42, 61, 72, 74]. However, GPUs lack native support for `mmap()`, presenting a significant hurdle, and fault-based on-demand paging may lead to higher costs because of CPU-GPU data transfer. Last, GPU apps' checkpoint images incur much more memory and storage costs. For example, the checkpoint image of MobileNet requires 468MB to store CPU data and 283MB to store GPU data. However, MobileNet has only 2.5M parameters, which requires less than 10MB of space if stored using float32.

To tackle the challenges, we propose the design of *on-demand and parallel restore* for GPU apps in gCROP. First, gCROP introduces a *delegation-based* parallel restore, breaking the *restorer barrier* by delegating the GPU recovery task to a global GPU Restore Server (an OS service). It leverages address space isolation as the new barrier, harmonizing the performance and feasibility. Second, gCROP supports on-demand restore by introducing `gmmap()`, a mechanism to support on-demand paging on GPU devices. gCROP combines both CPU `mmap()` and GPU `gmmap()` to mitigate recovery costs, and adopts a *profile-guided recovery* mechanism to reduce the costs caused by faults. Last, gCROP designs a *multi-checkpoint* mechanism, which increases the ratio of identical content among checkpoint images of different GPU apps. Based on the mechanism, gCROP can effectively reduce the storage costs through deduplication.

We have successfully built the prototype of gCROP, the first GPU runtime (for serverless and other scenarios) with low startup latency. We implement gCROP based on CRIU with AMD GPU, and evaluate it using both microbenchmarks and real-world applications. Results show significant improvement in startup latency, 6.4x–24.7x compared with booting from scratch and 3.9x–23.5x over the state-of-the-art method (CRIU).

2 MOTIVATION

2.1 GPU-based Serverless Computing

Serverless computing. Serverless computing or Function as a Service (FaaS) is a trending cloud computing paradigm. In this model, users only need to submit their function code to the cloud provider to handle requests and don't have to manage resources such as servers by themselves. Serverless typically provides the ability to automatically scale (or auto-scale) based on load. During scaling, the FaaS platform needs to initialize a new instance for handling requests — this process is called cold start. As the execution time of serverless functions is usually very short, reducing cold start latency is crucial for improving the performance of serverless platforms. Many optimizations have been proposed to reduce startup latency, including caching [36, 45, 53, 61, 64, 66, 77], serverless-customized container designs [11, 30], fork-based approaches [31, 42, 53, 61, 74] and Checkpoint/Restore [33, 70–72].

GPUs are becoming more important for serverless apps. Heterogeneous devices and accelerators (like GPUs) are becoming more important for serverless computing, especially considering AI/ML workloads like LLM (ChatGPT). In the following text, we use *GPU process* to refer to apps that use both CPU and GPU and *CPU process* to refer to apps that only use CPU. Prior academic works [32, 37, 41, 48, 52, 62, 63, 76, 78, 80] and cloud vendors [5, 65] have already attempted to integrate GPUs into serverless platforms. Similar to CPU serverless, users only need to upload function code and model files to the GPU serverless platform. When requests arrive, the platform starts the function and loads the model onto the GPU to process.

Specifically, the startup and the execution of a GPU process can be divided into four stages: (1) loading the framework (e.g., loading the Python interpreter, PyTorch framework), (2) loading the model, (3) transferring the model to the GPU and (4) executing the model. Stage (3) and (4) implicitly involve a lot of trigger-based initialization, like the creation of GPU contexts, the initialization of GPU-accelerated libraries, and the compilation of intermediate GPU kernel code to machine code. In the rest of the section, we use Python and PyTorch as the framework by default, nevertheless, our method is general and can support other frameworks (e.g., TVM [3]).

Cold start challenges for GPU apps. The startup latency is high for GPU-based serverless apps, and prior optimizations for CPU-based serverless apps are not feasible. First, GPU apps consume significantly more resources than regular CPU functions, making caching-based methods more costly. For example, a fully warmed-up MobileNet requires 792MB of memory, which is larger than most CPU-based serverless functions, and 322MB of GPU VRAM, which is much more expensive than regular memory. Second, GPU processes lack some characteristics of CPU processes, making certain optimizations (e.g., fork()) not directly applicable to GPU functions. For example, GPUs lack the OS support for fork(), and GPU memory usually does not support the copy-on-write feature, both of which are commonly used techniques for the rapid startup of CPU functions. Last, GPU functions have more startup procedures and more complex states, leading to longer initialization time compared to most CPU functions. Although some traditional optimization methods can still be applied, their effectiveness is limited.

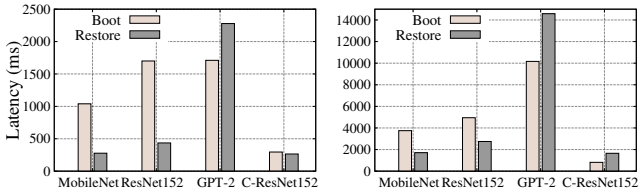
Table 1: The startup and execution latency breakdown. *C-ResNet152 is generated using TVM [3], while others are implemented on PyTorch. We conduct our tests on an AMD GPU (§6) and clear the page cache before each test.*

	MobileNet	ResNet152	GPT-2	C-ResNet152
Parameters	2.5M	60.2M	774.0M	60.2M
Load framework	PyTorch: 1,990.0ms			160.3ms
Load model	89.1ms	1,146.0ms	5,850.7ms	448.1ms
First transfer	869.6ms	928.0ms	1,317.2ms	196.2ms
(Second transfer)	(9.1ms)	(59.5ms)	(272.8ms)	(33.8ms)
First execution	802.0ms	882.6ms	965.4ms	13.1ms
(Second execution)	(5.8ms)	(16.3ms)	(255.5ms)	(10.5ms)
Total	3,750.7ms	4,946.6ms	10,123.3ms	817.7ms

2.2 A Quantitative Analysis on GPU Startup

Latency breakdown. We analyze the cost of cold start latency for common GPU serverless apps (Table 1). Specifically, we break down the startup and the execution latency for three apps with different sizes of models: MobileNet (2.4 million parameters) represents a small model, ResNet152 (60.2 million parameters) represents a medium model, and GPT-2-Large (774.0 million parameters) represents a large model. All these apps are based on Python and PyTorch. We also break down the latency of ResNet152 with TVM (based on C++) to show the impacts of different GPU frameworks. We obtain the transfer and execution latency by measuring the corresponding framework APIs or ROCm APIs. Because there is a lot of trigger-based initialization (referred to as *GPU runtime initialization*), which costs a lot of time during the first call to these APIs, we measure both transfer and execution latency twice. The "Second transfer" and "Second execution" in the table can represent the real transfer and execution costs after a model has been fully initialized and warmed up.

As shown in Table 1, during startup and the first execution, the GPU process spends a substantial amount of time on initialization and the actual execution time is small compared to the initialization latency. In the case of MobileNet, the actual execution time is only 0.15% of the total time, while



(a) Startup with cache. (b) Startup without cache.

Figure 1: Comparison between booting from scratch and restoring. ***(a) We cache the model files and the checkpoint image files in the page cache, and warm up the environment. (b) We clear the page cache and do not perform any warm-up operation.***

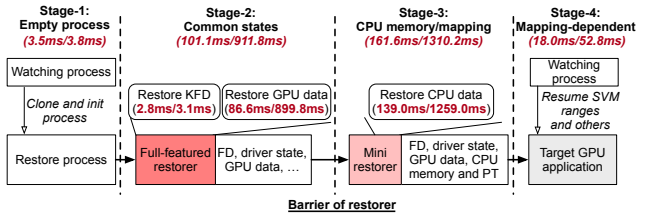


Figure 2: Breakdown of GPU restore. ***We label the latency of each stage for small and large GPU apps (MobileNet and GPT-2-Large). We test the latency with all image data cached in host memory.***

for GPT-2-Large, it is just 2.52%. Additionally, for small and medium models, loading framework takes the most significant cost, constituting 53% and 40% of the total costs. For large models, loading models takes the most significant costs, amounting to 57% of total costs.

State-of-the-art works. To mitigate the long startup latency of GPU apps, prior efforts have explored three types of optimizations. First, **skip the (almost) whole initialization with caching** [76, 78, 80]. A serverless platform can cache hot (or recently used) GPU apps on the CPU (host memory) and GPU (VRAM). When a request arrives and the cache is hit, the request can be directly handled by the GPU apps, achieving optimal performance with minimal startup latency (e.g., routing and queuing). However, the effectiveness and efficiency of caching highly rely on the policies, which are still challenging to design in real-world scenarios. Besides, caching may incur non-trivial resource overheads. An optimization is to only cache partial data to CPU and GPU, while loading others on-demand, called *partial caching* [55, 77], which can better balance the performance and the resource costs. We have implemented partial caching in GPU to only cache the framework, however, our results (§6) show that loading models and transferring them to the GPU still bring non-trivial costs, e.g., the latency for MobileNet is 243ms and for GPU-2-Large is 925ms.

Second, **skip the initialization of GPU runtime with API forwarding**, e.g., FaaSwap [79] and others [43, 60]. This method relies on a proxy-based design — a pre-initialized GPU server (directly communicating with GPUs) acts as a proxy and GPU applications send the CUDA [10]/ROCm [23] commands and the data to the proxy for processing. It can effectively optimize the GPU runtime initialization costs (about 800ms for a PyTorch application in a warmed environment on our AMD platform). However, it introduces a new type of cost — the inter-process communication (IPC) overhead between the GPU application and the GPU server. Besides, it still requires loading framework and models, which also take non-trivial costs for GPU apps. Furthermore, different GPU applications need to share the GPU servers, leading to potential security issues.

Last, due to limitations of caching whole apps and forwarding, some works explore **optimized loading with a set of techniques** [25, 44, 81]. For example, Safetensors [25] adopts mmap()-based loading. Although it can achieve better performance, it still causes significant page faults and cannot fully utilize multi-tier memory hierarchy [44]. ServerlessLLM [44] takes one step further, proposing a better image format for sequential and chunk-based loading (no serialization), and also utilizing a parallel design to effectively leverage multi-tier features. SAGE [81] re-organizes the startup procedure and attempts to decouple the initialization of the process and the preparation of the model data and process them simultaneously. However, all these approaches cannot optimize the process initialization costs (about 1600ms for a PyTorch application in a warmed environment on our platform, including about 800ms to load the framework and 800ms to initialize the GPU runtime), which is the primary overhead for applications with small models.

Initless-booting with Checkpoint/Restore (C/R). A promising approach to optimize the startup latency is to skip the initialization with states reusing, e.g., Checkpoint/Restore (C/R) or fork(), which has been a common method to optimize the cold start latency of CPU functions [33, 70–72]. It saves the states of an already initialized process into images via checkpointing and then restores the process states through replayable execution, which skips the initialization phase, thus speeding up process startup. CRIU [9] is currently one of the most popular Checkpoint/Restore tools. However, C/R is rarely explored for GPU serverless applications because of the lack of both OS and hardware support.

Recently, GPU vendors have started efforts to support C/R for GPU applications, e.g., CRIU AMDGPU plugin [28]. To understand the effectiveness, we adopt the method and present an evaluation for common GPU applications, as shown in Figure 1. As a result, we observe that C/R effectively optimizes

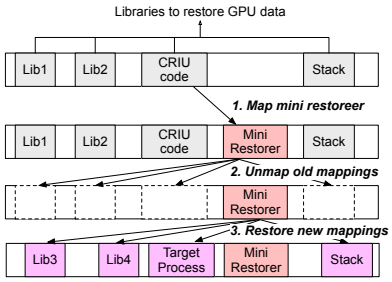


Figure 3: The memory mapping of the restore process during the restore of CPU and GPU data. *Step 1 takes place in Stage-2. Step 2 and Step 3 take place in Stage-3.*

the start latency for GPU applications with small models, e.g., MobileNet. However, the costs are still high, achieving several hundred milliseconds even in the case of data cached in page cache. For GPT-2-Large, restoring takes even longer compared to booting due to the larger amount of data that needs to be recovered.

Performance breakdown of GPU Checkpoint/Restore (C/R). We break down the restore phase of CRIU into four stages, as shown in Figure 2. First (**Stage-1**), CRIU processes will read the checkpoint image, perform basic initialization, and fork() a new process (called *restore process*). Initially the restore process can execute like a regular process and can be regarded as a *full-featured restorer*. The parent (CRIU process) is called *watching process*. Stage-1 takes small costs, about 3.5–3.8ms.

Second (**Stage-2**), the full-featured restorer will recover most of states and data except for CPU memory, including CPU file descriptors, GPU data, kernel driver states in AMD GPUs (called kernel fusion driver, KFD [28]), etc. Here we use *GPU data* to refer to memory contents that are managed by AMD GPU driver (most in VRAM, some in the host memory), and *CPU data* to refer to regular memory contents that need to be restored (all in the host memory). The costs of Stage-2 depend on the size of the model, e.g., MobileNet takes 101ms while GPT-2-Large takes 912ms.

Third (**Stage-3**), the restore process will restore the CPU memory and mappings (i.e., page tables). To achieve this, the restore process will unmap its old mappings (including CRIU code, libraries, etc.) and map new ones for the target process. To perform this operation, the restore process requires some code in memory (specifically, the code responsible for unmapping and restoring target memory). Therefore, CRIU installs and jumps to a *mini restorer* (Figure 3, Step 1), which includes self-contained code and data in a region not used by either the restore process or the target process. The mini restorer will unmap all other memory regions (Figure 3, Step 2) that may potentially overlap with the new memory mappings, and then recover the CPU memory states and mappings during this stage (Figure 3, Step 3). As the application usually maintains the whole model in the memory, the costs of Stage-3 also highly depend on the model size, i.e., MobileNet takes 162ms while GPT-2-Large takes 1310ms.

Last (**Stage-4**), some states, e.g., shared virtual memory (SVM) [14] ranges, are CPU mapping-dependent, which can only be recovered after Stage-3. As the restore process has no restorer anymore, AMD provides a new driver interface that allows the watching process to help the restore process recover these states. Consequently, the restore process becomes the target process capable of handling requests.

Storage overhead. Besides the restore latency, another challenge for GPU C/R is the high storage overhead. To checkpoint programs' states, CRIU will generate multiple image files, including metadata for the process states, CPU memory data, GPU device metadata, and GPU data. Among these, the CPU and GPU data images incur significant overhead. For example, the checkpoint image of MobileNet requires 468MB to store CPU data and 283MB to store GPU data, while the model parameters only require <10MB of space (if stored using float32). Therefore, to apply C/R for cold start optimization, it is crucial to minimize storage overhead.

2.3 Insights and Challenges

We present gCROP, a new GPU runtime system (based on Checkpoint/Restore) that achieves low startup latency with two insights. First, GPU data restore (Stage-2) and CPU data restore (Stage-3) are two major bottlenecks with no theoretical dependency. gCROP aims to restore the two parts in **parallel**, fully utilizing the CPU and PCIe (GPU) bandwidth to restore the states efficiently. Second, since not all restored states are utilized during execution [42], gCROP employs **on-demand** states restore to further optimize the latency. Furthermore, by simultaneously enabling our on-demand and parallel features, we can overlap computation and data restoration during the execution phase, thereby further reducing latency.

However, designing the on-demand and parallel restore for GPU apps is challenging.

Challenge-1: restorer barrier restricts the parallel restore of CPU and GPU data. An intuitive parallel method is to utilize a background thread to restore GPU data (Stage-2) when the main thread is restoring the CPU data (Stage-3). However, we observe there is a strong barrier between Stage-2 and Stage-3, called the *restorer barrier*, as shown in Figure 2, restricting the parallel design. Specifically, to restore the GPU data (Stage-2), the restore process utilizes various libraries for complex operations, such as employing libdrm for DMA operations (Figure 3). However, in Stage-3, to restore the target process's memory states, the mini restorer will unmap

all old mappings (Figure 3 Step 2), which means all these libraries will be unmapped, making it almost impossible to restore GPU data in this stage. Consequently, CRIU must first execute the GPU data recovery in Stage-2, followed by transitioning to the mini restorer for Stage-3 (Figure 3 Step 1) and unmapping the libraries, code, and data used in Stage-2 (Figure 3 Step 2). This sequential execution prevents the parallel processing of Stage-2 and Stage-3, leading to the establishment of the restorer barrier — a strict dependency between the two stages.

To address this challenge, we propose a *delegation*-based parallel restore mechanism. Rather than conducting all restoration within the restore process, we advocate transferring control of both the GPU memory regions that need to be restored and data to a global GPU restore service (designed in this work), delegating the restore of GPU data to this service. This approach allows GPU data to be restored in a separate address space, leveraging address space isolation as the new restorer barrier. Consequently, GPU and CPU data can be restored in parallel within distinct address spaces. We explain the details of this design in §3.2.

Challenge-2: GPUs bring new challenges for on-demand recovery. When optimizing Checkpoint/Restore for CPU processes, a common strategy involves utilizing mmap() for on-demand recovery [42, 61, 72, 74]. However, GPUs lack native support for mmap(), presenting a significant hurdle. What's worse, in GPU processes, both the CPU and GPU act as accessors for some memory regions managed by the GPU kernel driver [28], significantly heightening the complexity of on-demand restore. Moreover, as the GPU must traverse PCIe each time it accesses data not in VRAM, the overhead of on-demand restore surpasses that of mmap-based methods for CPU processes.

To tackle this challenge, we propose an on-demand solution based on multiple page tables for both CPUs and GPUs. Specifically, we introduce gmmap() (GPU mmap) to support on-demand GPU paging and combine mmap() and gmmap() to enable on-demand GPU recovery. Additionally, to mitigate potential performance issues arising from GPU page faults, we introduce a profile-guided approach to the background restore of GPU data using the GPU Restore Server, leveraging the knowledge of page access orders.

Challenge-3: checkpoint image consumes much space. To reduce storage overhead, a potential optimization is to deduplicate identical contents among checkpoint images of different GPU applications. Since these applications are often based on the same GPU runtime and framework, this approach may save a lot of storage space. However, we observe that due to diverse data layouts across processes, identical content is relatively rare among different images.

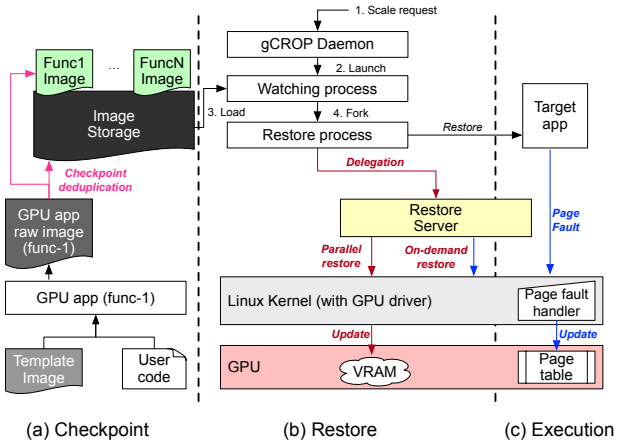


Figure 4: Overview of gCROP. *gCROP is a GPU runtime for low startup latency.*

To tackle this challenge, gCROP introduces *multi-checkpoint* mechanism, which can effectively increase the ratio of identical contents. Specifically, we observe that, unlike the booted processes which usually have diverse layouts, restored processes have the fixed mapping (as the saved ones). Therefore, unlike prior systems that checkpoint the app at the entry-point, gCROP will perform checkpoint twice — the first for the GPU framework and the second for the apps. All the apps using the same framework will share the same framework checkpoint, increasing the number of identical pages.

3 DESIGN

To overcome the above challenges, we propose gCROP (GPU Checkpoint/Restore made On-demand and Parallel). gCROP utilizes C/R technology to bypass the initialization phase of applications and optimizes the restore phase through two innovative approaches: parallel restore (§3.2) and on-demand restore (§3.3). In the restore phase, gCROP first parallelizes the restore of the process states and the GPU data. Then once the necessary data is restored, the process begins to execute, and any further data required is restored on demand or in the background. Moreover, to save space, we propose a multi-checkpoint mechanism (§3.4) to split the function image into its private parts and the common parts shared among functions and only store the private parts once. Next, we first introduce the overall architecture, and then explain the techniques.

3.1 System Overview

Platform. gCROP targets similar programming models of nowadays serverless platforms like AWS Lambda [4] and others [2]. gCROP requires developers to explicitly specify the hardware resources of the functions, e.g., the GPU type

and the requested size. Developers can write their functions based on a specific language runtime supported by the platform, and can utilize the GPU devices if specified. When a GPU-dependent function is uploaded to the platform, gCROP will prepare a checkpoint image for it and profile the GPU data restore order. When a request arrives, the API Gateway of a platform schedules a function's instance (when scaling) to machines satisfying the hardware requirements, and then notify the gCROP Daemon on each machine to load the checkpoint image of the function and restore the instance.

Architecture. Figure 4 shows the architecture of gCROP. gCROP includes four components: gCROP Daemon, Image Storage, Restore Server, and the kernel driver (for page fault handler). The entire workflow can be divided into four phases: checkpoint phase, profile phase, restore phase and execution phase. The first two phases are offline (performed during function uploading) and the others are online (triggered when requests arrive).

In the checkpoint phase (Figure 4-a), gCROP will boot an instance when a function is uploaded, and then checkpoint the target process into the raw checkpoint image, and deduplicate the raw image to save space (§3.4). When the gCROP Daemon receives a request (a new instance scheduled), it launches a CRIU watching process to restore the target process. To accelerate the restore phase, the restore of GPU data is delegated to Restore Server (§3.2), as shown in Figure 4-b. Moreover, Restore Server adopts an on-demand mechanism to restore the GPU data. It allows the restore process to proceed to the next phase without restoring all the GPU data and continues to restore the GPU data in the background (on-demand) (§3.3). In the execution phase (Figure 4-c), the page fault mechanism intercepts memory access requests in the GPU kernel for those unrestored areas. To mitigate the costs caused by page faults, gCROP will also profile the GPU apps offline to generate a trace, and utilize the trace to guide the restore procedure to recover pages that are highly possible used first (§3.3).

3.2 Parallel Restore

We first explain how to enable parallel restore for GPU apps, as shown in Figure 5. To break the restorer barrier, gCROP introduces two new primitives: delegation and asyncNotification, as well as a dedicated system service called Restore Server.

Restore Server. gCROP introduces a system service, Restore Server, to assist and boost the procedure of restore. Restore Server exposes a set of interfaces to allow CRIU watching processes (launched by gCROP Daemon) and restore processes to invoke, through IPC (domain sockets). Unlike the GPU servers used in prior API forwarding systems, Restore Server is only involved during the startup and does not intercept GPU APIs during execution. Therefore, it will not

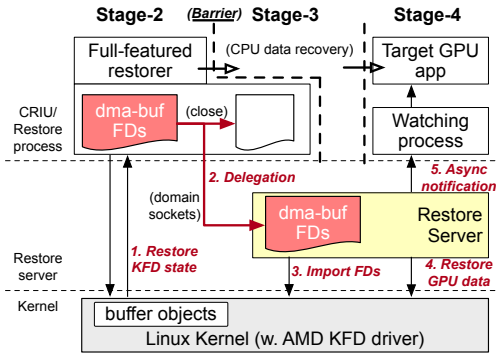


Figure 5: Parallel restore.

incur overhead to the application execution during runtime. In our prototype, we implement Restore Server in userspace.

GPU data management and hardware support. We first explain the mechanisms to manage GPU data in current GPU systems, which is the basics of our design. GPUs usually support low-level interfaces and abstractions to manage GPU data, which is necessary for both sharing memory and C/R. For example, AMD GPU and ROCm runtime manage the GPU data using the abstraction called *buffer objects*. A buffer object is a data structure used to manage memory (such as GPU VRAM) utilized by the GPU driver, and each buffer object represents a contiguous virtual memory area [50]. Besides, the AMD GPU driver provides two interfaces [1], import() and export(), to allow processes to export a buffer object to a Linux dma-buf [6], or import a buffer object from a dma-buf. The dma-buf is supported by Linux to share buffers for hardware (DMA) access across multiple device drivers and subsystems, and can be exposed as a file descriptor in userspace for applications to use.

Delegation. The key insight to enable parallel restore is to delegate the GPU data restore to Restore Server to break the restorer barrier. Specifically, when the restore phase begins, the restore process first restores the driver states through ioctl() interface and gets a set of dma-buf file descriptors (FDs) which represent the memory regions to be restored. Instead of importing these file descriptors in the restore process, the restore process will transfer the FDs to Restore Server at the beginning of Stage-2 through a Unix domain socket. It will also transfer the function image FD to Restore Server. After receiving these FDs, Restore Server imports them and obtains memory access permissions for these regions. This method separates the restore of CPU and GPU data into different address spaces, which breaks the restorer barrier. Consequently, Restore Server can restore GPU data in the background, allowing the restore process to continue with other tasks without waiting for the GPU data to be fully restored.

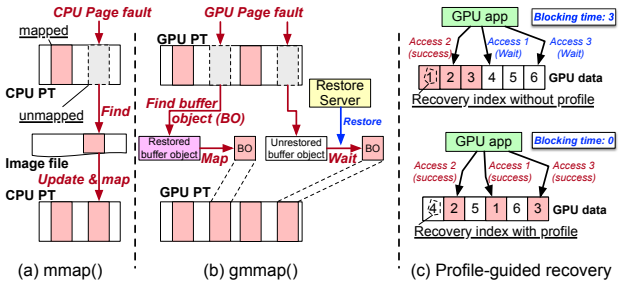


Figure 6: On-demand restore.

Async notification. As there are two parallel threads to perform the restore, we need to sync the progress at some point. In CRIU, one reasonable sync point is the beginning of Stage-4, which will restore the remaining states and start to execute. To enable the synchronization, when the restore process completes its work on CPU data restore, the watching process will check whether Restore Server has completed the GPU data restore. If not, it waits for the completion notification of Restore Server before allowing the restore process to execute.

The asynchronous notification will be further released with the design of on-demand restore (§3.3), in which the restore process only waits for data not applied for on-demand restore, while others can be recovered during runtime in an on-demand way.

Optimization with caching. In addition to its core functionalities, gCROP can leverage caching to further enhance performance. By employing various policies, Restore Server can proactively cache GPU data of specific (hot) functions in memory buffers (host memory or VRAM), thereby reducing the overhead associated with GPU data restore. This caching optimization is not easily achievable with existing CRIU systems, which lacks a global service for managing GPU data. Currently, CRIU can only cache relevant data in the Linux page cache before restoration, requiring an additional memory copy from the page cache to align memory buffers for CPU-GPU communication. The introduction of Restore Server empowers the system with caching capabilities tailored specifically for GPU C/R.

3.3 On-demand Restore

To further mitigate the startup latency, gCROP introduces the on-demand restore for GPU apps that can recover both CPU and GPU data during runtime. Our solution can be divided into three parts: a traditional mmap-based method for CPU data, a request interception mechanism based on GPU page fault mechanism for GPU data (called gmmap()), and a profile-guided recovery mechanism to mitigate the blocking costs caused by GPU page faults.

On-demand CPU data restore. We utilize the mmap() and on-demand paging supported by OS kernel as well as page faults by CPU to enable the on-demand CPU data restore, as shown in Figure 6-a. Specifically, there are two cases. First, during Stage-3, the restore process will load the CPU memory from the checkpoint image and restore the page mapping, which may bring significant costs as GPU apps usually hold large memory on the CPU side. To mitigate the costs, we follow prior efforts [42, 61, 71, 73], and try to use mmap() to map the checkpoint image directly to the memory. To this end, we modify the function image data reading part in CRIU to mmap the corresponding function image area. When the GPU app accesses unrestored memory, a page fault will be triggered and the data will be loaded on-demand, as shown in the figure.

Second, GPU pages may also be (directly) mapped to the CPU page tables [28], which are restored by Restore Server in the background. In this case, a CPU page fault still occurs when an access touches un-restored GPU pages. The OS kernel will redirect the fault to GPU driver, which will be handled by gCROP's fault handler. As the faulting data is GPU data, gCROP reuses the same logic for gmmap() to handle the case.

On-demand GPU data restore (gmmap()***).*** The basic idea of GPU on-demand paging is similar to CPU — intercepting memory accesses to unrecovered areas and retrying the accesses after the related area restoration. We observe that existing GPU vendors can support the required hardware primitives: GPU page faults and memory access retry (e.g., AMD XNACK feature [12]).

To achieve this, we delay the GPU page mapping and prepare the dedicated fault handler in the kernel. In the original CRIU, when the restore process sends an ioctl request to the kernel to restore the driver states, the GPU driver will reconstruct all buffer objects of the process and immediately map the areas represented by these buffer objects into the GPU page table. So the restored process can access the corresponding GPU memory areas. However, since our GPU data is prepared in parallel and on-demand, we must delay the mapping operation. Specifically, during restoration, gCROP will skip the mapping of on-demand buffer objects and mark them as not ready. To notify the driver when buffer objects are restored, gCROP's kernel driver provides a new ioctl interface to Restore Server to use.

In the running phase, when the app accesses the un-restored area, it will trigger a GPU page fault to our page fault handler, as shown in Figure 6-b. In this handler, gCROP checks whether the buffer object of the accessed address has been restored by Restore Server. If yes, gCROP's driver will update the GPU page table for the buffer object. If not, the system will wait until Restore Server calls the ioctl interface.

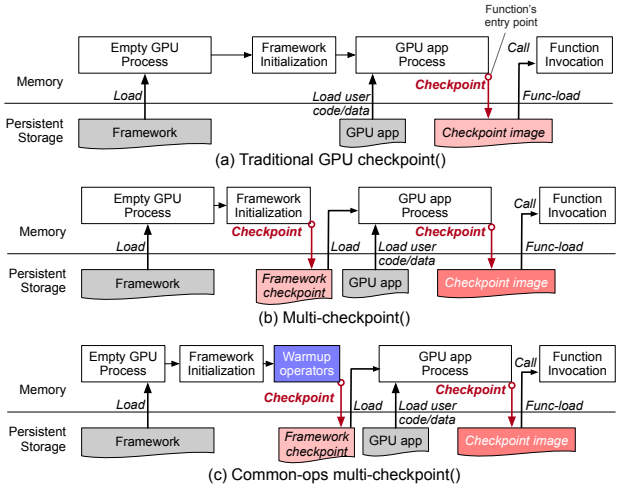


Figure 7: Multi-checkpoint mechanism.

We discuss two differences here. First, we perform restoration at the granularity of buffer objects. The main reason for the design choice is that GPU apps may access near pages in the same buffer object, recovering the whole object once can avoid future faults and handling. It is easy to extend the system to support other granularities like pages. Second, instead of fetching the requested pages directly, gCROP still relies on Restore Server to prepare the data. The main reason is to avoid costly and complex synchronization between the driver and the user-mode Restore Server. With our profile-guided recovery optimization, this design choice performs well in practice.

Profile-guided recovery mechanism. Because copying data from CPU to GPU memory is costly, we use a background restore approach managed by Restore Server instead of copying only when a page fault is triggered. However, in this method, the restore order is important. Poor restore orders may result in urgently needed data being restored last, causing unnecessary delays. We notice that for most models, the computation order is usually fixed [35, 49], which implies a fixed memory access sequence as well. So we propose an application-layer transparent profiling mechanism to get the restore order of GPU data, as shown in Figure 6-c. Specifically, we modify the AMD GPU page fault handler to export the sequence of buffer objects triggering page faults. This can be done offline. During the restore phase, Restore Server follows this sequence to restore the data.

3.4 Multi-Checkpoint Mechanism

To mitigate the storage costs caused by checkpoint images, gCROP presents the *multi-checkpoint mechanism*.

Idea of deduplication. The basic idea to resolve the costs is to share the content of checkpoint images among GPU apps. Specifically, as GPU apps usually depend on the same GPU runtime, frameworks and operators, their checkpoint files may have the same content that can be deduplicated. For example, two GPU apps running on PyTorch may share the same content for the Python interpreter and PyTorch libraries. Deduplicating the same content in the checkpoint images may save a lot of space.

Challenges. However, one significant challenge is that — even different apps share the same content, the data is not well-aligned in the same address (or offset) in the checkpoint images, and cannot be easily deduplicated. Figure 7-a shows a typical procedure of checkpoint. First, the framework (e.g., PyTorch) is loaded as the empty GPU process and performs initialization. Then the app loads the app-specific code and data, and is saved as a checkpoint image at the entry point. Nevertheless, different apps may have different data layouts on both the CPU and GPU sides, leading to a poor deduplication result.

Multi-checkpoint. We observe that, unlike the booted processes which usually have diverse layouts, restored processes have the fixed mapping (as the saved ones) and have a higher possibility to be deduplicated. Based on this observation, we propose *multi-checkpoint* — instead of performing a one-time checkpoint, we perform twice, as shown in Figure 7-b. Specifically, gCROP will perform a checkpoint after loading framework, generating a checkpoint image called *framework checkpoint*. The framework checkpoint only needs to be dumped once for each framework. Then different GPU apps are subsequently checkpointed based on the same framework checkpoint — gCROP restores from the framework checkpoint, loads the GPU app, and performs the second checkpoint. This only needs apps to specify a startup function and doesn't need to access the internals of apps. This method can significantly increase the identical regions among GPU apps. For example, the MobileNet application can have 22% identical regions compared to the framework checkpoint for CPU data.

Multi-checkpoint with warmed operators. Much of the initialization for GPU runtime occurs in the first time of model transfer and execution, which also contains non-trivial regions that can be deduplicated. However, with the above multi-checkpoint method, these regions may be missed. To further improve the benefits of deduplication, we introduce the second optimization, *warmed operators*, as shown in Figure 7-c. Specifically, gCROP will run some common GPU operators to initialize the GPU runtime (e.g., run torch.addmm to initialize rocBLAS [24]), after loading the framework and before dumping the framework checkpoint. gCROP allows platforms to determine the policies for warm operators, and

our results show that these warmed operators will not increase the final image size.

Deduplication procedure. In gCROP, we implement CPU data deduplication by comparing their data to the framework checkpoint and GPU data deduplication by comparing their data among different app checkpoints. For CPU data, we compare the template image and the function image at the granularity of pages, considering pages to be the same only if both the virtual address and the data are identical. For GPU data, we compare all images at the granularity of buffer objects, where buffer objects are treated identically if they have the same size and data, without requiring the virtual addresses to match. In this way, each function image only needs to store the parts that differ from the base image.

Co-design with restore. To accelerate the restore phase, the Restore Server can cache identical buffer objects (<200MB) in the framework checkpoint into the GPU memory before the restore phase. These buffer objects are almost identical to all the images. As a result, Restore Server can copy the data from the cached buffer object to the corresponding region using a VRAM-to-VRAM DMA transfer instead of a memory-to-VRAM DMA transfer.

4 IMPLEMENTATION

We implement gCROP based on the AMD GPU MI50, ROCm version 5.6.0, CRIU version 3.19, and Linux kernel 6.8.0.

Restore Server. We implement Restore Server as a system service, which will be launched and initialized during machine boot. In the restore phase, various restore processes and watching processes establish connections with the server through UNIX domain sockets. The server provides three interfaces: recv_driver_fd, recv_command, and restore_begin. Recv_driver_fd is used to receive the device file descriptor (FD) opened by the restore process. In later GPU data restore, Restore Server needs this FD to notify the kernel driver when a buffer object is restored. Driver can use this FD to determine which process the buffer object belongs to. Recv_command is used to receive the GPU data image FD, the dma-buf FD of the region to be restored, and the necessary metadata for restoration, such as whether it is on-demand, the size of the region to be restored, etc. Restore_begin indicates that the corresponding restore process has completed transmitting the command, and upon receiving this command, Restore Server will begin the restoration. We also provide a set of interfaces for communication and notification with the watching processes.

Restore Server supports various cache policies. In our implementation, gCROP can cache the corresponding data in memory buffers at the buffer object granularity and mark them using the st_dev (device ID) and st_ino (file serial number) of the corresponding file descriptor. During restoration, gCROP compares the st_dev and st_in of the received GPU data image file descriptor. If it has already been cached, the restoration starts directly from the memory buffer; otherwise, data is read from the corresponding file descriptor.

GPU page fault handling. In the process of handling a page fault, our page fault handler is divided into three steps. The first step is to find the corresponding buffer object based on the address that triggers the page fault. The second step is to check if the buffer object has completed restoration. For every buffer object, we use a bit in its flag to indicate whether the recovery is complete. The third step is to update the mapping with amdgpu_vm_update_range() and other function calls.

On-demand buffer object choose. Now we only apply our on-demand mechanism to buffer objects explicitly allocated by the apps (e.g., hipMalloc). Specifically, we modify the low-level GPU runtime (e.g., ROCm) to export the virtual address ranges of user-explicitly allocated memory (less than 20 lines) and then select the buffer objects matching these address ranges to serve as on-demand buffer objects. For other buffer objects, we restore them before program execution and don't apply the on-demand method to them.

Page cache management. The presence or absence of some dynamic libraries (such as the PyTorch library, rocBLAS library, etc.) in the page cache can influence performance. Therefore, to ensure the fairness of experiments, we modify vmtouch [16] to implement the checkpoint and restore functionality of the page cache. Specifically, we first run a fully warmed-up PyTorch process and record the cache status of various files in the page cache at that time (<600MB). This is achieved using the mincore() syscall. In our tests, we reproduce the page cache state according to the record and clear other page caches, which accelerates program execution and ensures that all evaluation are under the same page cache status.

5 DISCUSSION

Generality. gCROP requires modifications and support from low-level GPU software, e.g., driver and runtime. Due to Nvidia's closed-source stack, we implement our prototype on the AMD GPU platform. However, our method is not strictly dependent on the AMD GPU devices. As long as the following requirements are met, gCROP can be easily ported to other vendors.

First, GPU vendors should provide basic support to checkpoint and restore GPU-related kernel states. As GPU vendors usually maintain their own kernel-mode states, C/R systems like CRIU cannot directly save and recover a GPU-related process. Instead, CRIU relies on plugins to handle the device-specific states, e.g., AMDGPU plugin [28]. Recently, Nvidia

also starts the support for CRIU by providing a close-sourced plugin (in binary) [20]. We believe in the near future, the first requirement will become the basic feature of GPU devices. Second, GPU devices should provide a page fault mechanism to support on-demand restore. AMD supports page faults through its XNACK feature [12], which allows the GPU to retry memory accesses after a page fault when the accessed data does not exist in VRAM. Nvidia also supports the page fault mechanism to enable on-demand migration in unified virtual memory (UVM) [29]. Last, we need GPU memory transfer mechanism to support parallelized restore. AMD GPUs provide a set of interfaces (e.g., amdgpu_bo_export) to export and import the same buffer objects between different processes, which allows us to parallelize the restore procedure. For Nvidia, since its runtime API already supports IPC APIs (e.g., cudaIpcOpenMemHandle) to share the same GPU memory area, we believe it has low-level APIs that can directly interact with kernel drivers without the CUDA runtime to share GPU memory areas.

Besides, our insights also have a certain generality. For the first design, NVIDIA GPUs also tend to utilize CRIU for Checkpoint/Restore and will face similar challenges as AMD GPUs for parallel restore. For on-demand restore, the key optimization lies in the observation that the application does not require all data at startup. This characteristic comes from the application and does not change with the architecture. For the multi-checkpointing mechanism, the same content arises from using the same framework and GPU runtime, and all Checkpoint/Restore-based methods will encounter similar issues.

Multiple GPU support. Although our prototype is implemented on a single GPU, the design does not conflict with multiple GPU environments. In fact, gCROP may have more optimization opportunities in multiple GPU scenarios. For example, both AMD and Nvidia support direct GPU-to-GPU copies [13, 15] and one possible optimization is to use high-speed connections (e.g., NvLink [21]) between GPUs to accelerate GPU data recovery through direct GPU-to-GPU memory access. When an instance already exists on GPU 1 and a new identical app instance needs to be created on GPU 2, instead of reading data from disk or host memory, it can be faster to copy data directly from instance 1 to instance 2 via GPU-to-GPU memory access. These instances can delegate the corresponding GPU regions to Restore Server and then Restore Server can perform this operation. We leave the design and optimizations for multi-GPU C/R as future work.

Security. Since all function images are based on one template checkpoint, their positions of stack, heap, or libraries can be the same, which cannot be simply protected by the ASLR (Address Space Layout Randomization) technology. One solution is to only enable the multi-checkpoint mechanism in the scope of one tenant, or update the template image periodically. Cloud vendors can determine the policies to use the technique and balance the storage costs and security.

Another potential concern is that the multi-checkpoint mechanism requires application support, which may suggest cloud vendors require access to the internals of apps. However, since our method only needs apps to specify a startup function, developers can easily implement this on the client side by following the cloud vendors' guidelines.

Integration with container or virtual machine runtime. Currently, our work mainly focuses on optimizing the overhead of GPU app startup. In a serverless platform, user functions generally run in containers or virtual machines. gCROP and prior container or VM runtimes can be easily integrated, i.e., after an empty container/VM is launched, the system can invoke gCROP to restore a GPU app instead of booting from scratch.

Workload scope. Since our method requires checkpointing and profiling, it may be costly for applications that are only launched once in the cloud (i.e., non-repetitive). To avoid resource waste in these cases, platforms can allow developers to specify the basic pattern of their apps/workloads and only enable C/R when necessary.

6 EVALUATION

In the evaluation, we try to answer the following questions:

- How does gCROP reduce the startup and end-to-end latency of GPU applications? (§6.2)
- What is the end-to-end latency breakdown? (§6.3)
- How does our profile-guided recovery mechanism optimize the latency? (§6.4)
- How does the multi-checkpoint mechanism save the storage costs? (§6.5)
- How does gCROP reduce memory usage of applications? (§6.6)
- How do different cache configurations affect latency? (§6.7)
- How does gCROP perform when scaling multiple instances? (§6.8)

6.1 Experimental Setup

Testbed. We use an x86-64 server with an 8-core Intel i7-10700 CPU, 16GB memory and one AMD Radeon Instinct MI50 GPU (60 CUs and 16GB of memory) to evaluate. The OS is Ubuntu 18.04 with Linux kernel 6.8.0. All experiments are conducted in an official docker image with ROCm 5.6.0, PyTorch 1.12 and Ubuntu 20.04. The version of CRIU is 3.19.

Workloads. We use 7 representative pre-trained models (as shown in Table 2): MobileNet, ResNet18, Inception-V3,

Table 2: Workloads. ***"Warm exe." refers to the execution time of the model after it has been loaded into the GPU and completely warmed up. "Cold exe." refers to the total boot time from scratch.***

Models	Parameters	Warm exe.	Cold exe.
Hello	/	0.2ms	3.3s
MobileNet	2.5M	5.8ms	3.8s
ResNet18	11.7M	2.7ms	3.9s
Inception-V3	27.2M	12.1ms	4.4s
ResNet152	60.2M	16.3ms	5.0s
BERT-Base	109.5M	18.9ms	4.7s
OPT-350M	331.2M	154.0ms	5.6s
GPT-2-Large	774.0M	255.5ms	10.1s
C-ResNet152	60.2M	10.5ms	0.8s

ResNet152 from TorchVision [18] and BERT-Base, OPT-350M, GPT-2-Large from Transformers [75], covering the parameter range from 2.5 million to 774.0 million. The models are not modified. In addition to these models, we also select an app (called Hello) that performs only a single matrix addition and multiplication (torch.addmm) to approximate the scenario with zero model parameters (Table 2) and a ResNet152 app from DISB benchmark [47], which is generated by TVM [3] and based on the C++ environment (called C-ResNet152).

Similar to previous work [49], we use a synthetic dataset for all model inputs. All vision models use 224*224 RGB images. BERT-Base configures a sequence length of 512 while OPT-350M and GPT-2 configure 1024.

At the start of every experiment, we clear the page cache. Subsequently, we reconstruct the page cache according to our record (see §4 page cache management) and restore Hello app once to warm up the environment. In all experiments (except §6.7), similar to previous works [46, 49, 79] that cache all models in memory, we keep the model and image files in the page cache for our baselines and in Restore Server memory buffer for gCROP.

Baseline system. We compare gCROP to four baseline systems: booting from scratch, cold partial caching, warm partial caching and CRIU. Partial caching refers to preemptively keeping alive a Python interpreter that has loaded the PyTorch framework for applications except C-ResNet152. For C-ResNet152, partial caching refers to starting the process and pausing it before loading the model. Cold partial caching does not perform any additional warming on this basis. Warm partial caching builds on cold partial caching by loading and executing each model once, and then releasing the models. This allows the framework layer to be fully warmed up, so subsequent processes only need to load the respective models to transition to the related apps.

Since nearly all AI applications rely on a few frameworks, we believe that cold partial caching can represent the lower bound of partial caching methods for cold startup in GPU scenarios. For security reasons, cloud services may not allow different users' models to run in the same process to warm up once. In such cases, we can only perform warm-up by running some common models and operators, which would result in higher startup latency compared to our warm partial caching method. Therefore, we believe that warm partial caching can represent the upper bound of partial caching methods. CRIU denotes the baseline Checkpoint/Restore method.

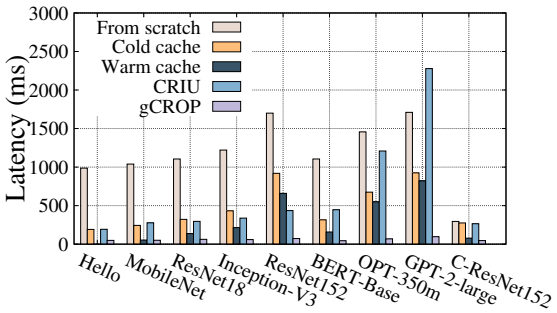


Figure 8: Startup latency. ***Cold cache represents cold partial caching and warm cache represents warm partial caching.***

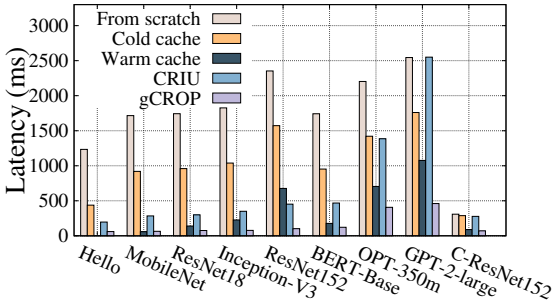
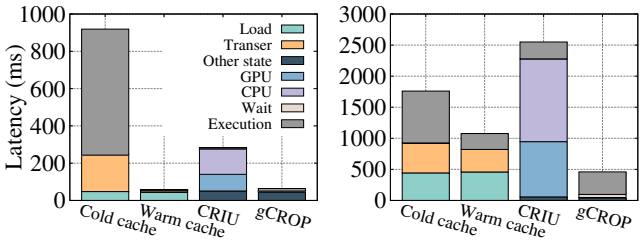


Figure 9: End-to-end latency.

6.2 Startup and End-to-end Latency

We first evaluate the startup and the end-to-end latency of gCROP with diverse applications to show its effectiveness.

Startup latency. Figure 8 shows the results. Except for Hello app and MobileNet, gCROP achieves the best performance in all other models. Specifically, gCROP can boost the startup latency of most models to 44ms–73ms, and for large models like GPT-2-Large, the startup latency is (firstly) reduced to 97ms. For the smallest model (MobileNet), gCROP's startup latency is comparable to warm partial caching but is 20.6x,



(a) MobileNet. **(b) GPT-2-Large.**

Figure 10: The end-to-end latency breakdown.

4.83x and 5.50x faster than booting from scratch, cold partial caching and CRIU, respectively. For the largest model (GPT-2-Large), gCROP improves startup latency by 17.6x, 9.5x, 8.4x, and 23.5x compared to booting from scratch, cold partial caching, warm partial caching, and CRIU, respectively. This result demonstrates that the improvement of gCROP is general for diverse models, ranging from a small model with 2.5 million parameters to a large model with 774.0 million parameters.

End-to-end latency. Figure 9 shows the results of end-to-end latency. Except for Hello app and MobileNet, gCROP achieves the best performance in all other models. Specifically, most models in gCROP only need 60ms–121ms to complete an end-to-end execution and OPT-350M and GPT-2-Large need 406ms and 460ms. Compared to cold partial caching and CRIU, gCROP is 14.4x and 4.4x faster for the smallest model and 3.8x and 5.5x for the largest model. For warm partial caching, gCROP is up to 6.6x faster for ResNet152 and 2.3x faster for GPT-2-Large. For MobileNet, gCROP is only slightly slower than warm partial caching and requires 63.9ms while warm partial caching needs 58.9ms, which is because the overhead of loading and transferring small models is not significant for warm partial caching, while gCROP needs to restore the entire process.

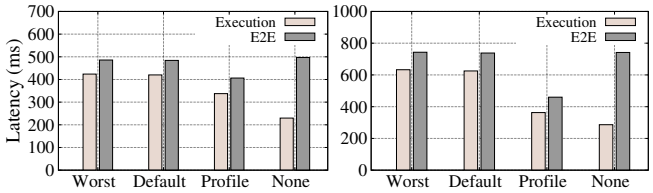
6.3 Latency Breakdown

In this section, we break down the latency of different systems with two models, the smallest model (MobileNet) and the largest model (GPT-2-Large), as shown in Figure 10. For booting from scratch, its latency is approximately the sum of the latency of cold partial caching and the overhead of loading the framework (about 800ms in this case). We do not draw this baseline in the figure. For cold partial caching and warm partial caching, we divide the end-to-end latency into three parts: model load time ("Load" in the figure), transfer time (Transfer), and execution time (Execution). Load time refers to the duration required to load the model from a file into host memory, while transfer time refers to the duration needed to move the model from host memory to GPU memory. For CRIU, the latency is divided into CPU data restore time (CPU), GPU data restore time (GPU), and other process state restore time (Other state). The CPU restore time refers to the duration required to restore CPU data in Stage-3, while the GPU restore time includes both the time needed to restore the GPU driver states and the GPU data in Stage-2. The remaining restore time is the other process state restore time. For gCROP, we add wait time (Wait), which represents the time the restore process waits in Stage-4 for Restore Server to recover buffer objects that are not applied on-demand. If Restore Server completes the recovery before the restore process enters Stage-4, this time is 0. Additionally, in gCROP, since GPU data restore is done in the background, GPU restore time refers to the time taken to restore the GPU driver states and the time required to transfer the file descriptors related to the on-demand buffer objects to Restore Server.

Comparison with cold partial caching. Since GPU runtime initialization costs a lot of time during the first execution in the cold partial caching method, gCROP is significantly faster regardless of whether the model is large or small. Even the execution time of the model alone in the cold partial caching method exceeds the entire restore and execution time of gCROP. Moreover, the cold partial caching method requires additional system memory to cache processes.

Comparison with warm partial caching. For MobileNet, because loading and transferring small models does not take much time for warm partial caching, the end-to-end time of gCROP is slightly longer than the warm partial caching method. However, as the model size increases, warm partial caching incurs significant overhead due to the need for deserialization during model loading. Additionally, since warm partial caching lacks on-demand technology, it cannot overlap computation with data transfer, further widening the performance gap with gCROP.

Comparison with CRIU. The improvements over CRIU mainly come from four aspects. First, in the restore phase, not all data in memory is needed to restore (e.g., the model parameters stored in memory). gCROP uses mmap to almost eliminate the overhead of restoring memory data (less than 2ms). Second, the parallel restore of GPU data and process states, combined with the restore of on-demand GPU data, hides most of the GPU memory restore time within the process state restore and execution time. For MobileNet, the non-overlapping GPU driver states and data restore time is only 3.5 ms, and for GPT-2-large, it is also only 53 ms. Third, GPU data is pre-cached in Restore Server, reducing the time CRIU takes to read GPU data from the page cache into the buffers. Fourth, by caching the same GPU data in GPU memory, the time spent on DMA transfers from system memory to GPU memory is reduced.



(a) OPT-350M. (b) GPT-2-Large.

Figure 11: Execution and end-to-end time of different on-demand GPU data recovery orders.

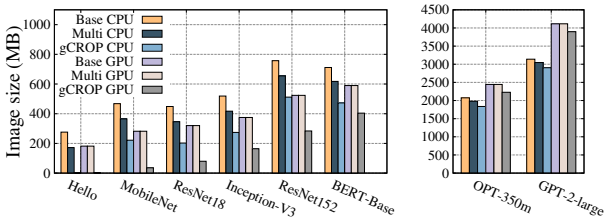
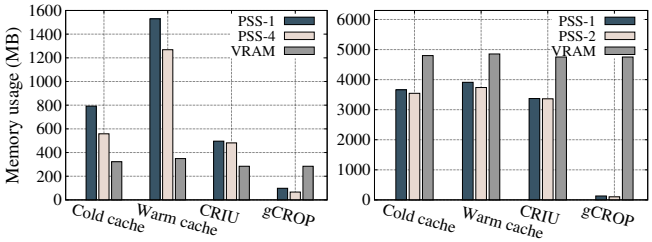


Figure 12: Image size of different checkpoint mechanisms.



(a) MobileNet. (b) GPT-2-Large.

Figure 13: Memory usage.

Comparison between small and large models. In gCROP, because mmap() is used, the CPU memory restore time almost does not increase with the model size. For the GPU part, the GPU driver state restore and FDs transfer time also almost do not increase with the model size. However, the GPU data restore time gradually becomes a bottleneck as the model size increases. For smaller models, the wait time is almost zero and the execution time is almost not affected by the on-demand mechanism. For larger models, because the restore of GPU data is slower than the restore of other process states, the target process must wait for it and then the execution time is longer than CRIU.

6.4 Profile-guided Recovery Mechanism

To understand the role of profile-guided recovery mechanism in optimizing execution latency and end-to-end (e2e) latency, we test the execution and e2e latency on GPT-2-Large and OPT-350M, as shown in Figure 11. We consider four representative cases. The **worst** case means the data that is needed first is restored last, representing a scenario where restoration is exactly in the worst order. The **default** case refers to the restoration following the default order without profiling according to CRIU. The **profile** case which gCROP does, involves the use of the profile-guided recovery mechanism. The **none** case means the GPU data on-demand mechanism is not utilized and all GPU data is restored in the restore phase.

Compared to the execution time of the none case, the increase of the execution time for the default case is 1.76x and 4.42x larger than that for the profile method for OPT-350M and GPT-2-Large. By using the profile method, the execution and the end-to-end latency are reduced by 1.24x and 1.19x for OPT-350M and 1.72x and 1.60x for GPT-2-Large compared to the default case. This demonstrates that this mechanism can effectively reduce the execution and end-to-end latency of large models by mitigating page faults. For small models, due to the short execution time and the short GPU data restore time, the effect of the profiling mechanism is not significant.

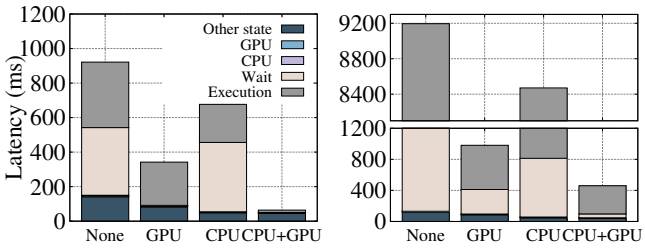
6.5 Benefits of Multi-Checkpoint

To show the benefits on resources by gCROP multi-checkpoint mechanism, we analyze the image sizes of models that are implemented on PyTorch for gCROP and the other two baselines (Figure 12). In our figure, the base method only checkpoints once and does no deduplication and the multi-checkpoint ("Multi" in the figure) method means to dump the framework checkpoint without warmed operators and only deduplicate by comparing the app checkpoint to the framework checkpoint.

Because all GPU apps depend on the same GPU runtime and AI framework, gCROP can save 8%–54% of CPU data image size and 5.3%–87.3% of GPU data image size for models (for Hello app gCROP can save 99%). For large models, because parameters occupy most of the CPU and GPU data, the optimization effect of this method is weaker. Additionally, because GPU is not used in the framework checkpoint stage, the framework checkpoint has no GPU data, and all GPU data need to be stored in their respective app images.

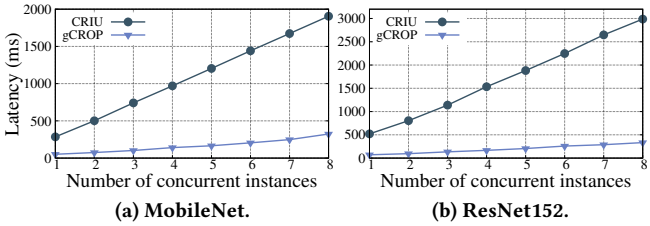
6.6 Memory Usage

Figure 13 compares memory usage of MobileNet and GPT-2-Large in all baselines and gCROP. We use proportional set size (PSS) to represent the memory usage, and VRAM size to represent the GPU memory usage. PSS-x means the average value of memory usage over x running processes. The average VRAM usage remains almost the same regardless of the number of processes, so we only show one. In this section,



(a) MobileNet. (b) GPT-2-Large.

Figure 14: End-to-end latency under different cache configurations.



(a) MobileNet. (b) ResNet152.

Figure 15: Total time of simultaneously scaling multiple instances. *Total time refers to the time from the start of the first instance to the completion of the last instance.*

we utilize CRIU-NS [8] to restore each process to its own namespace to avoid PID conflicts. Before measuring memory usage, all applications execute once to warm up after the restoration. We can see that gCROP achieves much lower PSS than all other baselines due to the on-demand restore of CPU data.

6.7 Different Cache Configurations

We test the startup latency of gCROP for the Mobilenet and GPT-2-Large models under different cache configurations, as shown in Figure 14. Specifically, we categorize the cache configurations into four types: cache no data (None), cache CPU data only (CPU), cache GPU data only (GPU), and cache all data (CPU+GPU). The CPU-only configuration stores the CPU data directly in the page cache, while the GPU-only configuration caches the GPU data in Restore Server.

For MobileNet, the startup latency increases by 10.8x, 9.1x, and 1.8x for the None configuration, the CPU-only configuration, and the GPU-only configuration compared to the CPU+GPU configuration, respectively, and the end-to-end latency increases by 14.4x, 10.6x, and 5.4x. For GPT-2-Large, the startup latency increases by 14.2x, 8.4x, and 4.2x for the none configuration, the CPU configuration, and the GPU configuration compared to the CPU+GPU configuration, respectively, and the end-to-end latency increases by 20.0x, 18.4x, and 2.1x. It can be observed that caching GPU data is crucial for the performance of gCROP. Furthermore, without caching GPU data, the restore of GPU data becomes the bottleneck in the restoration, regardless of the model size.

6.8 Concurrent Execution

Serverless requests often exhibit bursts, potentially requiring to simultaneously scale multiple instances. Even if model applications on the same GPU can increase throughput by enlarging the batch size instead of launching more instances, servers with multiple GPUs may still face situations where multiple instances need to scale simultaneously. We evaluate the startup time of gCROP for the case of simultaneously scaling multiple instances of MobileNet and ResNet152 (Figure 15). Similar to §6.6, we utilize CRIU-NS to restore each process to its own namespace to avoid PID conflicts. Additionally, we make some modifications to the CRIU invisible files restore logic to prevent conflicts when multiple CRIUs restore simultaneously.

In Figure 15, we evaluate the total latency with n (1–8) instances for simultaneous restore. Compared to CRIU, gCROP improves the total time by 5.5x–9.2x, with an average time of less than 40ms to complete a restoration of MobileNet and less than 45ms to complete a restoration of ResNet152.

7 RELATED WORK

Serverless inference systems. Industry such as AWS [65], Azure [59], and Alibaba [5], along with open-source systems like KServe [34], have provided support for serverless-based inference. Many academic works [32, 37, 48, 52, 62, 63, 76, 78, 80] attempt to build or optimize inference systems in serverless. However, these systems mainly focus on resource efficiency, scheduling or other issues, and skirt around the cold start issue or simply adopt caching based on various policies or predictions to avoid cold starts. In contrast, gCROP focuses on reducing the latency of cold start directly.

Cold-start optimizations in conventional model serving systems. There is a long line of research on the system support for model serving [35, 38–40, 46, 49, 51, 54, 58, 68]. Most of them are long-running, and load models into GPU memory in advance (before serving) and do not have startup problems. However, to serve models beyond GPU memory limits, some systems [35, 46, 49] introduce mechanisms to swap models between host and GPU. These mechanisms cause additional overheads when transferring a model from the host into the GPU, similar to the cold start issue in serverless systems. PipeSwitch [35] introduces the pipeline model transmission, while DeepPlan [49] proposes a direct-host-access mechanism to reduce waiting time for the model transfer. Despite these advancements in model transfer, these methods ignore the overhead of process initialization. Besides, they may not

satisfy the isolation requirements of serverless platforms. In contrast, gCROP considers the scenario of starting a complete inference process from scratch, achieving fast startup while meeting isolation requirements.

Keep-alive and pre-warming policy in serverless computing. Caching can completely or partially avoid the overhead of cold starts. To reduce the costs associated with caching, many works [36, 45, 53, 55, 57, 64, 66, 67, 77] focus on optimizing pre-warming or keep-alive policies. A common approach among these works is to dynamically determine the number and duration of caching instances based on historical data, aiming to minimize the number of cold starts while consuming fewer resources. Our work complements these techniques, as we focus on reducing the latency of a single startup. These various policies can be easily integrated with gCROP to further accelerate startup latency.

8 CONCLUSION

The long startup latency of GPU apps is known as the Achilles' Heel of today's model serving systems (based on serverless computing). To mitigate this issue, this paper presents gCROP, achieving <100ms startup latency for models with up to 774 million parameters. Although we are still far away from the ideal performance (e.g., <1ms in CPU apps), gCROP is a new milestone by combining a set of techniques to achieve the results. With more OS and hardware support, we believe future works can further reduce the latency.

ACKNOWLEDGMENTS

We sincerely thank our shepherd Dimitra Giantsidi and the anonymous SoCC'24 reviewers for their insightful suggestions. This work was supported in part by National Key R&D Program of China (2024QY1202), National Natural Science Foundation of China (No. 62302300), the Fundamental Research Funds for the Central Universities, Startup Fund for Young Faculty at SJTU (SFYF at SJTU), as well as research grants from Huawei Technologies.

REFERENCES

[1] 2024. AMD GPU DMA buffer. https://github.com/torvalds/linux/blob/v6.8/drivers/gpu/drm/amd/amdgpu/amdgpu_dma_buf.h. Accessed: 2024-07-04.
[2] 2024. Apache OpenWhisk is a serverless, open source cloud platform. http://openwhisk.apache.org/. Referenced 2024.
[3] 2024. Apache TVM. https://tvm.apache.org/. Accessed: 2024-07-04.
[4] 2024. AWS Lambda - Serverless Compute. https://aws.amazon.com/lambda/. Referenced Jan. 2024.
[5] 2024. Best practices for GPU-accelerated instances. https://www.alibabacloud.com/help/en/function-compute/latest/development-guide. Accessed: 2024-07-15.
[6] 2024. Buffer Sharing and Synchronization (dma-buf) — The Linux Kernel documentation. https://docs.kernel.org/driver-api/dma-buf.html. Accessed: 2024-07-04.
[7] 2024. Compute - Amazon EC2 Instance Types - AWS. https://aws.amazon.com/ec2/instance-types/. Accessed: 2024-07-04.
[8] 2024. CR in namespace - CRIU. https://criu.org/CR_in_namespace. Accessed: 2024-07-04.
[9] 2024. CRIU. https://criu.org/Main_Page. Accessed: 2024-07-04.
[10] 2024. CUDA Toolkit - Free Tools and Training | NVIDIA Developer. https://developer.nvidia.com/cuda-toolkit. Accessed: 2024-07-04.
[11] 2024. Google gVisor: Application Kernel for Containers. https://github.com/google/gvisor. Referenced 2024-07-16.
[12] 2024. GPU memory — ROCm Documentation. https://ROCm.docs.amd.com/en/latest/conceptual/gpu-memory.html. Accessed: 2024-07-04.
[13] 2024. GPUDirect | NVIDIA Developer. https://developer.nvidia.com/gpudirect. Accessed: 2024-07-04.
[14] 2024. Heterogeneous Memory Management (HMM) — The Linux Kernel documentation. https://www.kernel.org/doc/html/v5.0/vm/hmm.html. Accessed: 2024-07-04.
[15] 2024. HIP Runtime API Reference: PeerToPeer Device Memory Access — HIP 6.2.41134 Documentation. https://rocm.docs.amd.com/projects/HIP/en/latest/doxygen/html/group___peer_to_peer.html. Accessed: 2024-07-04.
[16] 2024. hoytech/vmtouch: Portable file system cache diagnostics and control. https://github.com/hoytech/vmtouch. Accessed: 2024-07-04.
[17] 2024. Introducing ChatGPT | OpenAI. https://openai.com/index/chatgpt/. Accessed: 2024-07-04.
[18] 2024. Models and pre-trained weights — Torchvision 0.18 documentation. https://PyTorch.org/vision/stable/models.html. Accessed: 2024-07-04.
[19] 2024. NVIDIA Tesla V100 | NVIDIA. https://www.nvidia.com/en-gb/data-center/tesla-v100/. Accessed: 2024-07-04.
[20] 2024. NVIDIA/cuda-checkpoint: CUDA checkpoint and restore utility. https://github.com/NVIDIA/cuda-checkpoint. Accessed: 2024-07-04.
[21] 2024. NVLink & NVSwitch: Fastest HPC Data Center Platform | NVIDIA. https://www.nvidia.com/en-us/data-center/nvlink/. Accessed: 2024-07-04.
[22] 2024. opencontainers/runc. https://github.com/opencontainers/runc.git. Referenced 2024-07-15.
[23] 2024. ROCm Software. https://www.amd.com/en/products/software/rocm.html. Accessed: 2024-07-04.
[24] 2024. ROCm/rocBLAS: Next generation BLAS implementation for ROCm platform. https://github.com/ROCm/rocBLAS. Accessed: 2024-07-04.
[25] 2024. Safetensors: ML Safer for All. https://github.com/huggingface/safetensors. Accessed: 2024-07-04.
[26] 2024. [Snaps] Full snapshot + restore, firecracker-microvm/firecracker. https://github.com/firecracker-microvm/firecracker/issues/1184. Referenced April 2024.
[27] 2024. Sora | OpenAI. https://openai.com/index/sora/. Accessed: 2024-07-04.
[28] 2024. Supporting ROCm with CRIU. https://github.com/checkpoint-restore/criu/tree/criu-dev/plugins/amdgpu. Accessed: 2024-07-04.
[29] 2024. Unified Memory for CUDA Beginners | NVIDIA Technical Blog. https://developer.nvidia.com/blog/unified-memory-cuda-beginners/. Accessed: 2024-07-04.
[30] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX symposium on networked systems design and implementation (NSDI 20)*. 419–434.
[31] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. 2018. *SAND*: Towards High-Performance Serverless Computing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 923–935.

[32] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. 2020. BATCH: Machine Learning Inference Serving on Serverless Platforms with Adaptive Batching. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15. https://doi.org/10.1109/SC41405.2020.00073

[33] Lixiang Ao, George Porter, and Geoffrey M. Voelker. 2022. FaaSnap: FaaS made fast using snapshot-based VMs. In *Proceedings of the Seventeenth European Conference on Computer Systems* (Rennes, France) *(EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 730–746. https://doi.org/10.1145/3492321.3524270

[34] The KServe Authors. 2023. KServe. https://github.com/kserve/kserve. Accessed on 2024-06-22.

[35] Zhihao Bai, Zhen Zhang, Yibo Zhu, and Xin Jin. 2020. PipeSwitch: Fast Pipelined Context Switching for Deep Learning Applications. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 499–514. https://www.usenix.org/conference/osdi20/presentation/bai

[36] Vivek M. Bhasi, Jashwant Raj Gunasekaran, Prashanth Thinakaran, Cyan Subhra Mishra, Mahmut Taylan Kandemir, and Chita Das. 2021. Kraken: Adaptive Container Provisioning for Deploying Dynamic DAGs in Serverless Platforms. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SoCC '21)*. Association for Computing Machinery, New York, NY, USA, 153–167. https://doi.org/10.1145/3472883.3486992

[37] Seungbeom Choi, Sunho Lee, Yeonjae Kim, Jongse Park, Youngjin Kwon, and Jaehyuk Huh. 2022. Serving Heterogeneous Machine Learning Models on Multi-GPU Servers with Spatio-Temporal Sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 199–216. https://www.usenix.org/conference/atc22/presentation/choi-seungbeom

[38] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. 2017. Clipper: A Low-Latency Online Prediction Serving System. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 613–627. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/crankshaw

[39] Weihao Cui, Han Zhao, Quan Chen, Hao Wei, Zirui Li, Deze Zeng, Chao Li, and Minyi Guo. 2022. DVABatch: Diversity-aware Multi-Entry Multi-Exit Batching for Efficient Processing of DNN Services on GPUs. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 183–198. https://www.usenix.org/conference/atc22/presentation/cui

[40] Aditya Dhakal, Sameer G Kulkarni, and K. K. Ramakrishnan. 2020. GSLICE: controlled spatial sharing of GPUs for a scalable inference platform. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) *(SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 492–506. https://doi.org/10.1145/3419111.3421284

[41] Dong Du, Qingyuan Liu, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. 2022. Serverless computing on heterogeneous computers. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 797–813.

[42] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 2020. Catalyzer: Sub-Millisecond Startup for Serverless Computing with Initialization-Less Booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 467–481. https://doi.org/10.1145/3373376.3378512

[43] Henrique Fingler, Zhiting Zhu, Esther Yoon, Zhipeng Jia, Emmett Witchel, and Christopher J. Rossbach. 2022. DGSF: Disaggregated GPUs for Serverless Functions. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 739–750. https://doi.org/10.1109/IPDPS53621.2022.00077

[44] Yao Fu, Leyang Xue, Yeqi Huang, Andrei-Octavian Brabete, Dmitrii Ustiugov, Yuvraj Patel, and Luo Mai. 2024. ServerlessLLM: Low-Latency Serverless Inference for Large Language Models. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*. USENIX Association, Santa Clara, CA, 135–153. https://www.usenix.org/conference/osdi24/presentation/fu

[45] Alexander Fuerst and Prateek Sharma. 2021. FaasCache: Keeping Serverless Computing Alive with Greedy-Dual Caching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS 2021)*. Association for Computing Machinery, New York, NY, USA, 386–400. https://doi.org/10.1145/3445814.3446757

[46] Arpan Gujarati, Reza Karimi, Safya Alzayat, Wei Hao, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. 2020. Serving DNNs like Clockwork: Performance Predictability from the Bottom Up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 443–462. https://www.usenix.org/conference/osdi20/presentation/gujarati

[47] Mingcong Han, Hanze Zhang, Rong Chen, and Haibo Chen. 2022. Microsecond-scale Preemption for Concurrent GPU-accelerated DNN Inferences. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 539–558. https://www.usenix.org/conference/osdi22/presentation/han

[48] Vatche Isahagian, Vinod Muthusamy, and Aleksander Slominski. 2017. Serving Deep Learning Models in a Serverless Platform. *2018 IEEE International Conference on Cloud Engineering (IC2E)* (2017), 257–262. https://api.semanticscholar.org/CorpusID:21724528

[49] Jinwoo Jeong, Seungsu Baek, and Jeongseob Ahn. 2023. Fast and Efficient Model Serving Using Multi-GPUs with Direct-Host-Access. In *Proceedings of the Eighteenth European Conference on Computer Systems* (Rome, Italy) *(EuroSys '23)*. Association for Computing Machinery, New York, NY, USA, 249–265. https://doi.org/10.1145/3552326.3567508

[50] Jens Kehne, Jonathan Metter, and Frank Bellosa. 2015. GPUswap: Enabling Oversubscription of GPU Memory through Transparent Swapping. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Istanbul, Turkey) *(VEE '15)*. Association for Computing Machinery, New York, NY, USA, 65–77. https://doi.org/10.1145/2731186.2731192

[51] Tan N. Le, Xiao Sun, Mosharaf Chowdhury, and Zhenhua Liu. 2020. AlloX: compute allocation in hybrid clusters. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 31, 16 pages. https://doi.org/10.1145/3342195.3387547

[52] Jie Li, Laiping Zhao, Yanan Yang, Kunlin Zhan, and Keqiu Li. 2022. Tetris: Memory-efficient Serverless Inference through Tensor Sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA. https://www.usenix.org/conference/atc22/presentation/li-jie

[53] Zijun Li, Linsong Guo, Quan Chen, Jiagan Cheng, Chuhao Xu, Deze Zeng, Zhuo Song, Tao Ma, Yong Yang, Chao Li, and Minyi Guo. 2022. Help Rather Than Recycle: Alleviating Cold Startup in Serverless Computing Through Inter-Function Container Sharing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 69–84. https://www.usenix.org/conference/atc22/presentation/li-zijun-help

[54] Zhuohan Li, Lianmin Zheng, Yinmin Zhong, Vincent Liu, Ying Sheng, Xin Jin, Yanping Huang, Zhifeng Chen, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. AlpaServe: Statistical Multiplexing with Model Parallelism for Deep Learning Serving. In *17th USENIX Symposium on*

Operating Systems Design and Implementation (OSDI 23). USENIX Association, Boston, MA, 663–679. https://www.usenix.org/conference/osdi23/presentation/li-zhouhan

[55] Zhen Lin, Kao-Feng Hsieh, Yu Sun, Seunghee Shin, and Hui Lu. 2021. FlashCube: Fast Provisioning of Serverless Functions with Streamlined Container Runtimes. In *Proceedings of the 11th Workshop on Programming Languages and Operating Systems* (Virtual Event, Germany) *(PLOS '21)*. Association for Computing Machinery, New York, NY, USA, 38–45. https://doi.org/10.1145/3477113.3487273

[56] Qingyuan Liu, Dong Du, Yubin Xia, Ping Zhang, and Haibo Chen. 2023. The Gap Between Serverless Research and Real-world Systems. In *Proceedings of the 2023 ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '23)*. Association for Computing Machinery, New York, NY, USA, 475–485. https://doi.org/10.1145/3620678.3624785

[57] Qingyuan Liu, Yanning Yang, Dong Du, Yubin Xia, Ping Zhang, Jia Feng, James R. Larus, and Haibo Chen. 2024. Harmonizing Efficiency and Practicability: Optimizing Resource Utilization in Serverless Computing with Jiagu. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. USENIX Association, Santa Clara, CA, 1–17. https://www.usenix.org/conference/atc24/presentation/liu-qingyuan

[58] Zihan Liu, Jingwen Leng, Zhihui Zhang, Quan Chen, Chao Li, and Minyi Guo. 2022. VELTAIR: towards high-performance multi-tenant deep learning services via adaptive compilation and scheduling. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 388–401. https://doi.org/10.1145/3503222.3507752

[59] Microsoft. 2023. Azure ML. https://learn.microsoft.com/en-us/azure/machine-learning. Accessed on 2024-06-22.

[60] Diana M. Naranjo, Sebastián Risco, Carlos de Alfonso, Alfonso Pérez, Ignacio Blanquer, and Germán Moltó. 2020. Accelerated serverless computing based on GPU virtualization. *J. Parallel Distributed Comput.* 139 (2020), 32–42. https://api.semanticscholar.org/CorpusID:213589452

[61] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. *SOCK*: Rapid Task Provisioning with Serverless-Optimized Containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. 57–70.

[62] Qiangyu Pei, Yongjie Yuan, Haichuan Hu, Qiong Chen, and Fangming Liu. 2023. AsyFunc: A High-Performance and Resource-Efficient Serverless Inference System via Asymmetric Functions. In *Proceedings of the 2023 ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '23)*. Association for Computing Machinery, New York, NY, USA, 324–340. https://doi.org/10.1145/3620678.3624664

[63] Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. 2021. INFaaS: Automated Model-less Inference Serving. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 397–411. https://www.usenix.org/conference/atc21/presentation/romero

[64] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. 2022. IceBreaker: warming serverless functions better with heterogeneity. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 753–767. https://doi.org/10.1145/3503222.3507750

[65] AWS SageMaker. 2023. Machine Learning Service - Amazon SageMaker. https://aws.amazon.com/pm/sagemaker/.

[66] Divyanshu Saxena, Tao Ji, Arjun Singhvi, Junaid Khalid, and Aditya Akella. 2022. Memory deduplication for serverless computing with Medes. In *Proceedings of the Seventeenth European Conference on Computer Systems* (Rennes, France) *(EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 714–729. https://doi.org/10.1145/3492321.3524272

[67] Mohammad Shahrad, Rodrigo Fonseca, Íñigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 205–218.

[68] Haichen Shen, Lequn Chen, Yuchen Jin, Liangyu Zhao, Bingyu Kong, Matthai Philipose, Arvind Krishnamurthy, and Ravi Sundaram. 2019. Nexus: a GPU cluster engine for accelerating DNN-based video analysis. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 322–337. https://doi.org/10.1145/3341301.3359658

[69] Simon Shillaker and Peter Pietzuch. 2020. Faasm: lightweight isolation for efficient stateful serverless computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 419–433.

[70] Paulo Silva, Daniel Fireman, and Thiago Emmanuel Pereira. 2020. Prebaking Functions to Warm the Serverless Cold Start. In *Proceedings of the 21st International Middleware Conference* (Delft, Netherlands) *(Middleware '20)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3423211.3425682

[71] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. 2021. Benchmarking, Analysis, and Optimization of Serverless Function Snapshots. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS 2021)*. Association for Computing Machinery, New York, NY, USA, 559–572. https://doi.org/10.1145/3445814.3446714

[72] Kai-Ting Amy Wang, Rayson Ho, and Peng Wu. 2019. Replayable execution optimized for page sharing for a managed runtime environment. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–16.

[73] Xingda Wei, Fangming Lu, Rong Chen, and Haibo Chen. 2022. KRCORE: A Microsecond-scale RDMA Control Plane for Elastic Computing. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 121–136. https://www.usenix.org/conference/atc22/presentation/wei

[74] Xingda Wei, Fangming Lu, Tianxia Wang, Jinyu Gu, Yuhan Yang, Rong Chen, and Haibo Chen. 2023. No Provisioned Concurrency: Fast RDMA-codesigned Remote Fork for Serverless Computing. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 497–517. https://www.usenix.org/conference/osdi23/presentation/wei-rdma

[75] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander Rush. 2020. Transformers: State-of-the-Art Natural Language Processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, Qun Liu and David Schlangen (Eds.). Association for Computational Linguistics, Online, 38–45. https://doi.org/10.18653/v1/2020.emnlp-demos.6

[76] Yanan Yang, Laiping Zhao, Yiming Li, Huanyu Zhang, Jie Li, Mingyang Zhao, Xingzhen Chen, and Keqiu Li. 2022. INFless: a native serverless system for low-latency, high-throughput inference. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 768–781. https://doi.org/10.1145/3503222.3507709

[77] Hanfei Yu, Rohan Basu Roy, Christian Fontenot, Devesh Tiwari, Jian Li, Hong Zhang, Hao Wang, and Seung-Jong Park. 2024. RainbowCake: Mitigating Cold-starts in Serverless with Layer-wise Container

Caching and Sharing. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 335–350. https://doi.org/10.1145/3617232.3624871

[78] Minchen Yu, Zhifeng Jiang, Hok Chun Ng, Wei Wang, Ruichuan Chen, and Bo Li. 2021. Gillis: Serving Large Neural Networks in Serverless Functions with Automatic Model Partitioning. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*. 138–148. https://doi.org/10.1109/ICDCS51616.2021.00022

[79] Minchen Yu, Ao Wang, Dong Chen, Haoxuan Yu, Xiaonan Luo, Zhuohao Li, Wei Wang, Ruichuan Chen, Dapeng Nie, and Haoran Yang. 2024. FaaSwap: SLO-Aware, GPU-Efficient Serverless Inference via Model Swapping. arXiv:2306.03622 [cs.DC] https://arxiv.org/abs/2306.03622

[80] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. 2019. MArk: exploiting cloud services for cost-effective, SLO-aware machine learning inference serving. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference* (Renton, WA, USA) *(USENIX ATC '19)*. USENIX Association, USA, 1049–1062.

[81] Han Zhao, Weihao Cui, Quan Chen, Shulai Zhang, Zijun Li, Jingwen Leng, Chao Li, Deze Zeng, and Minyi Guo. 2024. Towards Fast Setup and High Throughput of GPU Serverless Computing. arXiv:2404.14691 [cs.DC] https://arxiv.org/abs/2404.14691